José Luiz Fiadeiro
Peter D. Mosses
Fernando Orejas (Eds.)

LNCS 3423

Recent Trends in Algebraic Development Techniques

17th International Workshop, WADT 2004
Barcelona, Spain, March 2004
Revised Selected Papers

Springer

José Luiz Fiadeiro Peter D. Mosses
Fernando Orejas (Eds.)

Recent Trends in Algebraic Development Techniques

17th International Workshop, WADT 2004
Barcelona, Spain, March 27-29, 2004
Revised Selected Papers

Springer

Volume Editors

José Luiz Fiadeiro
University of Leicester, Department of Computer Science
Leicester LE1 7RH, UK
E-mail: jose@fiadeiro.org

Peter D. Mosses
University of Wales Swansea, Department of Computer Science
Singleton Park, Swansea SA2 8PP, UK
E-mail: P.D.Mosses@swan.ac.uk

Fernando Orejas
Universitat Politècnica de Catalunya
Departament de Llenguatges i Sistemes Informàtics
Campus Nord C5, Jordi Girona 1-3, 08034 Barcelona, Spain
E-mail: orejas@lsi.upc.es

Library of Congress Control Number: 2005922176

CR Subject Classification (1998): F.3.1, F.4, D.2.1, I.1

ISSN 0302-9743
ISBN 3-540-25327-0 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11407355 06/3142 5 4 3 2 1 0

Lecture Notes in Computer Science 3423

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Editorial Board

Preface

This volume contains selected papers from WADT 2004, the 17th International Workshop on Algebraic Development Techniques. Like its predecessors, WADT 2004 focussed on the algebraic approach to the specification and development of systems, an area that was born around the algebraic specification of abstract data types and encompasses today the formal design of software systems, new specification frameworks and a wide range of application areas.

WADT 2004 took place at the Technical University of Catalonia (UPC), Barcelona, Spain, on 27–29 March 2004, and was organized by Fernando Orejas and Jordi Cortadella.

The program consisted of invited talks by Luís Caires (Universidade Nova de Lisboa, Portugal) and Reiko Heckel (University of Paderborn, Germany), and 33 presentations describing ongoing research on main topics of the workshop: formal methods for system development, specification languages and methods, systems and techniques for reasoning about specifications, specification development systems, methods and techniques for concurrent, distributed and mobile systems, and algebraic and co-algebraic foundations.

The Steering Committee of WADT, consisting of Michel Bidoit, José Fiadeiro, Hans-Jörg Kreowski, Peter Mosses, Fernando Orejas, Francesco Parisi-Presicce, and Andrzej Tarlecki, with the additional help of Christine Choppy and Till Mossakowski, selected several presentations and invited their authors to submit a full paper for possible inclusion in this volume. All submissions underwent a careful refereeing process. We are extremely grateful to all the referees who helped in reviewing the submissions: H. Baumeister, L. Caires, A. Cherchago, R. Heckel, R. Hennicker, F. Jacquemard, R. Klempien-Hinrichs, C. Lüth, S. Merz, W. Pawlowski, and L. Schröder.

This volume contains the final versions of the 14 contributions that were accepted. It contains also the invited paper of Reiko Heckel, co-authored with Sebastian Thöne.

The workshop was jointly organized with IFIP WG 1.3 (Foundations of System Specification), and received generous sponsorship from the following organizations:

- Spanish Ministry of Science and Technology (MCYT)
- Catalan Department for University, Research and Information Society (DURSI)
- Technical University of Catalonia (UPC)

David Banyeres, Robert Clariso, Kyller Costa, Nilesh Modi, Jiangtao Meng, Nikos Mylonakis, Sonia Perez, Edelmira Pasarella, and Elvira Pino provided invaluable help throughout the preparation and organization of the workshop. We are grateful to Springer for its helpful collaboration and quick publication.

Finally, we would like to announce that, starting in 2005, WADT will join forces and reputations with CMCS, the International Workshop on Coalgebraic Methods in Computer Science, to create a new high-level biennial international event: CALCO, the Conference on Algebra and Coalgebra in Computer Science.

December 2004 José Fiadeiro, Peter Mosses, Fernando Orejas

Lecture Notes in Computer Science

For information about Vols. 1–3331

please contact your bookseller or Springer

Vol. 3452: F. Baader, A. Voronkov (Eds.), Logic for Programming, Artificial Intelligence, and Reasoning. XI, 562 pages. 2005. (Subseries LNAI).

Vol. 3448: G.R. Raidl, J. Gottlieb (Eds.), Evolutionary Computation in Combinatorial Optimization. XI, 271 pages. 2005.

Vol. 3436: B. Bouyssounouse, J. Sifakis (Eds.), Embedded Systems Design. XV, 492 pages. 2005.

Vol. 3432: M. Beigl, P. Lukowicz (Eds.), Systems Aspects in Organic and Pervasive Computing - ARCS 2005. X, 265 pages. 2005.

Vol. 3423: J.L. Fiadeiro, P.D. Mosses, F. Orejas (Eds.), Recent Trends in Algebraic Development Techniques. VIII, 271 pages. 2005.

Vol. 3419: B. Faltings, A. Petcu, F. Fages, F. Rossi (Eds.), Constraint Satisfaction and Constraint Logic Programming. X, 217 pages. 2005. (Subseries LNAI).

Vol. 3418: U. Brandes, T. Erlebach (Eds.), Network Analysis. XII, 471 pages. 2005.

Vol. 3416: M. Böhlen, J. Gamper, W. Polasek, M.A. Wimmer (Eds.), E-Government: Towards Electronic Democracy. XIII, 311 pages. 2005. (Subseries LNAI).

Vol. 3415: P. Davidsson, B. Logan, K. Takadama (Eds.), Multi-Agent and Multi-Agent-Based Simulation. X, 265 pages. 2005. (Subseries LNAI).

Vol. 3414: M. Morari, L. Thiele (Eds.), Hybrid Systems: Computation and Control. XII, 684 pages. 2005.

Vol. 3412: X. Franch, D. Port (Eds.), COTS-Based Software Systems. XVI, 312 pages. 2005.

Vol. 3411: S.H. Myaeng, M. Zhou, K.-F. Wong, H.-J. Zhang (Eds.), Information Retrieval Technology. XIII, 337 pages. 2005.

Vol. 3410: C.A. Coello Coello, A. Hernández Aguirre, E. Zitzler (Eds.), Evolutionary Multi-Criterion Optimization. XVI, 912 pages. 2005.

Vol. 3409: N. Guelfi, G. Reggio, A. Romanovsky (Eds.), Scientific Engineering of Distributed Java Applications. X, 127 pages. 2005.

Vol. 3408: D.E. Losada, J.M. Fernández-Luna (Eds.), Advances in Information Retrieval. XVII, 572 pages. 2005.

Vol. 3407: Z. Liu, K. Araki (Eds.), Theoretical Aspects of Computing - ICTAC 2004. XIV, 562 pages. 2005.

Vol. 3406: A. Gelbukh (Ed.), Computational Linguistics and Intelligent Text Processing. XVII, 829 pages. 2005.

Vol. 3404: V. Diekert, B. Durand (Eds.), STACS 2005. XVI, 706 pages. 2005.

Vol. 3403: B. Ganter, R. Godin (Eds.), Formal Concept Analysis. XI, 419 pages. 2005. (Subseries LNAI).

Vol. 3401: Z. Li, L.G. Vulkov, J. Waśniewski (Eds.), Numerical Analysis and Its Applications. XIII, 630 pages. 2005.

Vol. 3398: D.-K. Baik (Ed.), Systems Modeling and Simulation: Theory and Applications. XIV, 733 pages. 2005. (Subseries LNAI).

Vol. 3397: T.G. Kim (Ed.), Artificial Intelligence and Simulation. XV, 711 pages. 2005. (Subseries LNAI).

Vol. 3396: R.M. van Eijk, M.-P. Huget, F. Dignum (Eds.), Agent Communication. X, 261 pages. 2005. (Subseries LNAI).

Vol. 3395: J. Grabowski, B. Nielsen (Eds.), Formal Approaches to Software Testing. X, 225 pages. 2005.

Vol. 3394: D. Kudenko, D. Kazakov, E. Alonso (Eds.), Adaptive Agents and Multi-Agent Systems III. VIII, 313 pages. 2005. (Subseries LNAI).

Vol. 3393: H.-J. Kreowski, U. Montanari, F. Orejas, G. Rozenberg, G. Taentzer (Eds.), Formal Methods in Software and Systems Modeling. XXVII, 413 pages. 2005.

Vol. 3391: C. Kim (Ed.), Information Networking. XVII, 936 pages. 2005.

Vol. 3390: R. Choren, A. Garcia, C. Lucena, A. Romanovsky (Eds.), Software Engineering for Multi-Agent Systems III. XII, 291 pages. 2005.

Vol. 3389: P. Van Roy (Ed.), Multiparadigm Programming in Mozart/OZ. XV, 329 pages. 2005.

Vol. 3388: J. Lagergren (Ed.), Comparative Genomics. VIII, 133 pages. 2005. (Subseries LNBI).

Vol. 3387: J. Cardoso, A. Sheth (Eds.), Semantic Web Services and Web Process Composition. VIII, 147 pages. 2005.

Vol. 3386: S. Vaudenay (Ed.), Public Key Cryptography - PKC 2005. IX, 436 pages. 2005.

Vol. 3385: R. Cousot (Ed.), Verification, Model Checking, and Abstract Interpretation. XII, 483 pages. 2005.

Vol. 3383: J. Pach (Ed.), Graph Drawing. XII, 536 pages. 2005.

Vol. 3382: J. Odell, P. Giorgini, J.P. Müller (Eds.), Agent-Oriented Software Engineering V. X, 239 pages. 2005.

Vol. 3381: P. Vojtáš, M. Bieliková, B. Charron-Bost, O. Sýkora (Eds.), SOFSEM 2005: Theory and Practice of Computer Science. XV, 448 pages. 2005.

Vol. 3379: M. Hemmje, C. Niederee, T. Risse (Eds.), From Integrated Publication and Information Systems to Information and Knowledge Environments. XXIV, 321 pages. 2005.

Vol. 3378: J. Kilian (Ed.), Theory of Cryptography. XII, 621 pages. 2005.

Vol. 3377: B. Goethals, A. Siebes (Eds.), Knowledge Discovery in Inductive Databases. VII, 190 pages. 2005.

Vol. 3376: A. Menezes (Ed.), Topics in Cryptology – CT-RSA 2005. X, 385 pages. 2005.

Vol. 3375: M.A. Marsan, G. Bianchi, M. Listanti, M. Meo (Eds.), Quality of Service in Multiservice IP Networks. XIII, 656 pages. 2005.

Vol. 3374: D. Weyns, H.V.D. Parunak, F. Michel (Eds.), Environments for Multi-Agent Systems. X, 279 pages. 2005. (Subseries LNAI).

Vol. 3372: C. Bussler, V. Tannen, I. Fundulaki (Eds.), Semantic Web and Databases. X, 227 pages. 2005.

Vol. 3371: M.W. Barley, N. Kasabov (Eds.), Intelligent Agents and Multi-Agent Systems. X, 329 pages. 2005. (Subseries LNAI).

Vol. 3370: A. Konagaya, K. Satou (Eds.), Grid Computing in Life Science. X, 188 pages. 2005. (Subseries LNBI).

Vol. 3369: V.R. Benjamins, P. Casanovas, J. Breuker, A. Gangemi (Eds.), Law and the Semantic Web. XII, 249 pages. 2005. (Subseries LNAI).

Vol. 3368: L. Paletta, J.K. Tsotsos, E. Rome, G.W. Humphreys (Eds.), Attention and Performance in Computational Vision. VIII, 231 pages. 2005.

Vol. 3367: W.S. Ng, B.C. Ooi, A. Ouksel, C. Sartori (Eds.), Databases, Information Systems, and Peer-to-Peer Computing. X, 231 pages. 2005.

Vol. 3366: I. Rahwan, P. Moraitis, C. Reed (Eds.), Argumentation in Multi-Agent Systems. XII, 263 pages. 2005. (Subseries LNAI).

Vol. 3365: G. Mauri, G. Păun, M.J. Pérez-Jiménez, G. Rozenberg, A. Salomaa (Eds.), Membrane Computing. IX, 415 pages. 2005.

Vol. 3363: T. Eiter, L. Libkin (Eds.), Database Theory - ICDT 2005. XI, 413 pages. 2004.

Vol. 3362: G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, T. Muntean (Eds.), Construction and Analysis of Safe, Secure, and Interoperable Smart Devices. IX, 257 pages. 2005.

Vol. 3361: S. Bengio, H. Bourlard (Eds.), Machine Learning for Multimodal Interaction. XII, 362 pages. 2005.

Vol. 3360: S. Spaccapietra, E. Bertino, S. Jajodia, R. King, D. McLeod, M.E. Orlowska, L. Strous (Eds.), Journal on Data Semantics II. XI, 223 pages. 2005.

Vol. 3359: G. Grieser, Y. Tanaka (Eds.), Intuitive Human Interfaces for Organizing and Accessing Intellectual Assets. XIV, 257 pages. 2005. (Subseries LNAI).

Vol. 3358: J. Cao, L.T. Yang, M. Guo, F. Lau (Eds.), Parallel and Distributed Processing and Applications. XXIV, 1058 pages. 2004.

Vol. 3357: H. Handschuh, M.A. Hasan (Eds.), Selected Areas in Cryptography. XI, 354 pages. 2004.

Vol. 3356: G. Das, V.P. Gulati (Eds.), Intelligent Information Technology. XII, 428 pages. 2004.

Vol. 3355: R. Murray-Smith, R. Shorten (Eds.), Switching and Learning in Feedback Systems. X, 343 pages. 2005.

Vol. 3354: M. Margenstern (Ed.), Machines, Computations, and Universality. VIII, 329 pages. 2005.

Vol. 3353: J. Hromkovič, M. Nagl, B. Westfechtel (Eds.), Graph-Theoretic Concepts in Computer Science. XI, 404 pages. 2004.

Vol. 3352: C. Blundo, S. Cimato (Eds.), Security in Communication Networks. XI, 381 pages. 2005.

Vol. 3351: G. Persiano, R. Solis-Oba (Eds.), Approximation and Online Algorithms. VIII, 295 pages. 2005.

Vol. 3350: M. Hermenegildo, D. Cabeza (Eds.), Practical Aspects of Declarative Languages. VIII, 269 pages. 2005.

Vol. 3349: B.M. Chapman (Ed.), Shared Memory Parallel Programming with Open MP. X, 149 pages. 2005.

Vol. 3348: A. Canteaut, K. Viswanathan (Eds.), Progress in Cryptology - INDOCRYPT 2004. XIV, 431 pages. 2004.

Vol. 3347: R.K. Ghosh, H. Mohanty (Eds.), Distributed Computing and Internet Technology. XX, 472 pages. 2004.

Vol. 3346: R.H. Bordini, M. Dastani, J. Dix, A.E.F. Seghrouchni (Eds.), Programming Multi-Agent Systems. XIV, 249 pages. 2005. (Subseries LNAI).

Vol. 3345: Y. Cai (Ed.), Ambient Intelligence for Scientific Discovery. XII, 311 pages. 2005. (Subseries LNAI).

Vol. 3344: J. Malenfant, B.M. Østvold (Eds.), Object-Oriented Technology. ECOOP 2004 Workshop Reader. VIII, 215 pages. 2005.

Vol. 3343: C. Freksa, M. Knauff, B. Krieg-Brückner, B. Nebel, T. Barkowsky (Eds.), Spatial Cognition IV. Reasoning, Action, and Interaction. XIII, 519 pages. 2005. (Subseries LNAI).

Vol. 3342: E. Şahin, W.M. Spears (Eds.), Swarm Robotics. IX, 175 pages. 2005.

Vol. 3341: R. Fleischer, G. Trippen (Eds.), Algorithms and Computation. XVII, 935 pages. 2004.

Vol. 3340: C.S. Calude, E. Calude, M.J. Dinneen (Eds.), Developments in Language Theory. XI, 431 pages. 2004.

Vol. 3339: G.I. Webb, X. Yu (Eds.), AI 2004: Advances in Artificial Intelligence. XXII, 1272 pages. 2004. (Subseries LNAI).

Vol. 3338: S.Z. Li, J. Lai, T. Tan, G. Feng, Y. Wang (Eds.), Advances in Biometric Person Authentication. XVIII, 699 pages. 2004.

Vol. 3337: J.M. Barreiro, F. Martin-Sanchez, V. Maojo, F. Sanz (Eds.), Biological and Medical Data Analysis. XI, 508 pages. 2004.

Vol. 3336: D. Karagiannis, U. Reimer (Eds.), Practical Aspects of Knowledge Management. X, 523 pages. 2004. (Subseries LNAI).

Vol. 3335: M. Malek, M. Reitenspieß, J. Kaiser (Eds.), Service Availability. X, 213 pages. 2005.

Vol. 3334: Z. Chen, H. Chen, Q. Miao, Y. Fu, E. Fox, E.-p. Lim (Eds.), Digital Libraries: International Collaboration and Cross-Fertilization. XX, 690 pages. 2004.

Vol. 3333: K. Aizawa, Y. Nakamura, S. Satoh (Eds.), Advances in Multimedia Information Processing - PCM 2004, Part III. XXXV, 785 pages. 2004.

Vol. 3332: K. Aizawa, Y. Nakamura, S. Satoh (Eds.), Advances in Multimedia Information Processing - PCM 2004, Part II. XXXVI, 1051 pages. 2004.

Table of Contents

Invited Technical Paper

Contributed Papers

Behavior-Preserving Refinement Relations Between Dynamic Software Architectures

Reiko Heckel[1] and Sebastian Thöne[2]

[1] Department of Computer Science
[2] International Graduate School Dynamic Intelligent Systems,
University of Paderborn, Germany
{reiko, seb}@upb.de

Abstract. In this paper, we address the refinement of abstract architectural models into more platform-specific representations. For each level of abstraction, we employ an architectural style covering structural restrictions on component configurations as well as supported communication and reconfiguration operations. Architectural styles are formalized as graph transformation systems with graph transformation rules defining the available operations. Architectural models are given as graphs to which one can directly apply the transformation rules in order to simulate operations and their effects.

In addition to previous work, we include process descriptions into our architectural models in order to control the communication and reconfiguration behavior of the components. The execution semantics of these processes is also covered by graph transformation systems.

We propose a notion of refinement which requires the preservation of both structure and behavior at the lower level of abstraction. Based on formal refinement relationships between abstract and platform-specific styles, we can use model checking techniques to verify that abstract scenarios can also be realized in the platform-specific architecture.

1 Introduction

In the development of complex software systems, a model of the *software architecture* [30] allows for early reasoning on the system at a high level of abstraction. An architectural model covers the involved run-time *configuration* of system components, the *communication* between these components, and possible *reconfiguration* operations that enable the system to react to upcoming requirements and events. Such *dynamic* architectures gain increasing attention in the context of e-business, self-healing, and mobile systems.

Since software architectures[1] are intended to bridge the gap between system requirements and implementation, they have to conform to both business-driven requirements as well as restrictions and mechanisms imposed by the chosen run-time infrastructure. In order to integrate both aspects, we propose a stepwise refinement approach starting with an abstract, *business-level* architecture which

[1] We use the term *software architecture* as a synonym for the *model* of an architecture.

J.L. Fiadeiro, P. Mosses, and F. Orejas (Eds.): WADT 2004, LNCS 3423, pp. 1–27, 2005.

can be derived from user and business requirements. This business-level architecture is then refined into a more concrete description which also integrates *platform-specific* aspects like supported reconfiguration operations and communication mechanisms.

A recent example of this general principle of model refinement is the *Model-Driven Architecture (MDA)* [26] put forward by the OMG. Here, platform-specific details are initially ignored at the model-level to allow for maximum portability. Then, these platform-independent models are refined by adding details required to map to a given target platform. At each refinement level, more assumptions on the resources, constraints, and services of the chosen platform are incorporated into the model.

Similarly, as described in a previous paper [3], we use *architectural styles* [2], formalized as graph transformation systems, for defining the assumptions on a certain level of platform abstraction, i.e., the vocabulary, structural constraints, and available communication and reconfiguration mechanisms. Then, an architecture at a certain level of abstraction has to conform to the corresponding architectural style.

In our previous work, we applied the available style-specific reconfiguration and communication operations to an architecture without further control. In this paper (see Section 4), we provide an extension which allows the definition of *processes* and their operational semantics. These processes control the order in which available operations are invoked by the individual software components. This leads to a more detailed picture of the architectural behavior.

We do not consider architectural refinement as the internal decomposition of components into subcomponents, as done by other authors, but rather focus on porting an abstract architecture to a more platform-specific level which usually requires additional platform-related entities and resources. For this purpose, we define different architectural styles for different levels of platform abstraction, namely a generic, platform-independent style for business-level architectures and a more specific style for architectural models at the platform-specific level.

When refining software architectures from the abstract to the concrete level, we have to preserve both structural and behavioral properties. This leads to the following two requirements:

1. **Architectural consistency:** After being ported to the lower level of abstraction, the concrete architecture has to satisfy the same functional requirements as the abstract architecture. Therefore, we have to refine configurations of components, connections, and other resources in a way that all business-relevant entities of the abstract architecture are also preserved at the concrete level.
2. **Behavior preservation:** Similarly, the concrete architecture has to preserve the abstract communication and reconfiguration behavior. In particular, we require that all business-relevant scenarios of the abstract architecture are also realizable in the concrete architecture.

While porting the abstract behavior to the platform-specific level we have to respect the capabilities of the chosen target platform according to its reconfigu-

ration and communication mechanisms. In many cases, depending on the current situation where an operation is to be applied and the effects of preceding actions, the refinement of an abstract action varies from other situations and cannot be decided locally. Thus, we believe that behavior preservation cannot be solved by a fixed syntactic mapping between abstract and concrete operations but has to be dealt with at a semantic level.

Further requirements include a high degree of *reusability* which means that the refinement relationship between certain levels of platform abstraction should not only apply for one specific system, but should be reusable for other architectures as well.

Since refinement is not an easy task and thus error-prone and cost-intensive if done by hand, we are also aiming at *tool support*. However, it is difficult to automate the *construction* of refined architectures, because this is a creative process, and computers cannot invent details for the concrete level that are not existent at the abstract level. Nevertheless, we intend to investigate tool support for *checking* if a concrete architecture satisfies the formal refinement relationship we prescribe for the refinement from abstract to concrete level. Combined with user interaction for modifying invalid concrete models, we achieve a semi-automated approach for creating refined architectural models.

The refinement relationship, as already proposed in [5], is *style-based* meaning that it is defined between two architectural styles rather than between individual architectures. Since this relationship can be applied to any instances of the styles, we achieve the desired degree of reusability.

To check for architectural consistency, we have to compare the business-relevant entities of the abstract and the concrete model. For this purpose, we use an *abstraction function* which lifts concrete models to the abstract style. To check for behavior preservation, we have to prove that all states of an abstract scenario are also reachable in a corresponding concrete scenario, preferably with the help of model checking techniques. For this purpose, we employ a contravariant *translation function* which transforms abstract states into requirements for states at the platform-specific level. A model checker can then search for concrete states satisfying these requirements.

The rest of this paper is organized as follows. We survey related work in Section 2. In Section 3, we revisit the modeling of architectural styles based on graph theory, and in Section 4 we extend the proposed architecture description technique by processes for controlling architectural behavior. In Section 5, we use this formal framework to define our notion of refinement under the obligation of architectural consistency, and Section 6 covers the problem of behavior preservation by a semantic requirement that can be checked by model checking tools. Section 7 concludes the paper.

2 Related Work

Refinement is a long-known design principle in software engineering. First ideas in the context of program development go back to Wirth [34]. In the sense of

a systematic top-down methodology, he argued for the expansion of high-level program instructions to lower level macros and procedures.

While Wirth mainly investigated sequential programs, the refinement of concurrent systems became popular as *action refinement* in the context of process algebras (cf. [17] for a survey on this topic). This field considers the refinement of abstract actions into sequences of concrete actions, also called *processes*, and the potential interleaving of multiple concurrent processes.

Our approach is different from this work for two reasons. First, we want to avoid a fixed, sometimes even syntactically defined substitution of an abstract action by a concrete process wherever the abstract action occurs. Instead, we are aiming at a more flexible notion of refinement which also allows for alternate refinements of an action depending on the context where the action occurs. Second, we also want to enable refinement in those cases where the two levels of abstraction are so different that it becomes hard to relate the corresponding actions with each other.

Apart from action refinement, we also have to mention the different notions of refinement in the field of software architecture. For instance, Batory et. al. [6] consider *feature refinement* which is modifying models, code, and other artifacts in order to integrate additional features with every refinement step. Different to this work, Canal et. al. [9] consider refinement as the decomposition of a software component into subcomponents and the specialization of components under certain compatibility conditions.

In our case, we neither want to add any extra-functionality to the architecture nor to look into the internals of the components, but we rather want to port a business-level architecture to a more platform-specific level considering all the restrictions and mechanisms of the chosen target platform.

Refinement of architectures in this sense has first been discussed by Moriconi et al. in [25]. Building a formalization in first-order logic, the authors describe a general approach of rule-based refinement replacing a structural pattern in an abstract style by its realization in the concrete style. The approach is related to ours, but focuses on refinement of the structure only and does not take reconfiguration and communication behavior into account. Also, applying the logic-based theory to concrete architecture description languages is not trivial. The general idea of rule-based refinement, however, is applicable in our context, too.

Garlan [16] stresses the fact that it is more powerful to have rules operating on architectural styles rather than on style instances. He formalizes refinements as abstraction functions from the concrete to the abstract style. We use a similar approach to define refinement relationships (see Section 5). Also, he argues that no single definition of refinement can be provided, but that one should state what properties are preserved. In our case, we concentrate on the preservation of architectural consistency and the dynamic semantics of reconfiguration and communication scenarios.

Other proposals on architecture refinement like [1, 12] concentrate on structural refinements only, which is complementary to our work. The only formal approach we are aware of that considers refinement of dynamic reconfiguration

can be found in [8]. But, the paper only sketches the ideas without any concrete definition. Moreover, the approach is targeted on the translation from one Architecture Description Language to another rather than on the refinement between architectural styles that represent different levels of platform abstraction.

Since we use graph transformation systems as the underlying formalism to describe dynamic software architectures, which is in the tradition of [21, 22, 24, 31, 33], it is also worth to look at existing work on refinement of graph transformation systems. The general idea is to relate the transformation rules and, thus, the behavior of an abstract graph transformation system to the rules of a more concrete transformation system. One can judge these refinement relationships along a continuum from syntactical relationships to more semantical ones.

Große-Rhode et. al. [18], for instance, propose a refinement relationship between abstract and concrete rules that can be checked syntactically. One of the conditions requires that, e.g., the abstract rule and its refinement must have the same pre- and post-conditions except for retyping. Based on this very restrictive definition they can prove that the application of the concrete rule expression yields the same behavior as the corresponding abstract rule. The draw-back of this approach is that it cannot handle those cases where the refining rule expression should have additional effects on platform-specific elements that do not occur in the abstract rule. And, similar to action refinement, the approach does not allow alternate refinements for the same abstract rule.

Similarly, the work by Heckel et. al. [20] is based on a syntactical relationship between two graph transformation systems. Although this approach is less restrictive as it allows additional (platform-specific) elements at the concrete level, it is still difficult to apply if there are no direct correspondences between abstract and concrete rules. Moreover, their objective is to project any given concrete transformation behavior to the abstract level and not vice versa.

In our work, we propose a more flexible, semantic-based notion of refinement. We do not define a fixed mapping between the various transformation rules but only between the structural parts of the graph transformation system. Then, we check whether all system states of an abstract model are also reachable at the concrete level, no matter by which order of transformation rules. By avoiding the functional refinement mapping between transformation rules, we can also relate transformation systems with completely different behavior, and we are flexible enough to cope with alternate refinements.

3 Graph Transformation Systems as Architectural Styles

As already introduced in [3], we use *architectural styles* as conceptual platform models. Such a platform model has to define the vocabulary of elements to be considered, to restrict the possible relationships among those elements, and to specify communication as well as reconfiguration mechanisms supported by the platform. We use different styles for different levels of platform abstraction.

In this section, we present the formal definition of architectural styles as *typed graph transformation systems* [10] together with two exemplary styles, namely

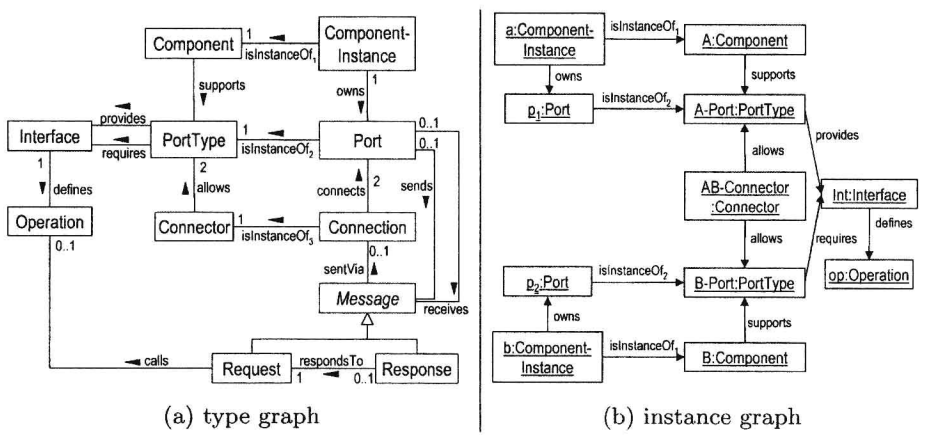
an abstract style for business-level architectures and a platform-specific style for service-oriented architectures. In Section 5, we explain how a refinement relationship between these styles can be used to refine business-level architectures, which abstract from platform-specific vocabulary and restrictions, to service-oriented architectures.

Informally, a typed graph transformation system consists of (1) a *type graph* to define the vocabulary of architectural elements, (2) a set of *constraints* to further restrict the valid models, and (3) a set of *graph transformation rules* for communication and reconfiguration operations. A system architecture that conforms to a given style is represented as an *instance graph* of the type graph.

Definition 1 (Graph and Graph Morphism). *A graph is a tuple* $G = (N, E, src, tar)$ *with a set* N *of nodes, a set* E *of edges, and functions* $src, tar : E \to N$ *that assign source and target nodes to each edge. A graph morphism* $f = (f_N, f_E) : G \to G'$ *is a pair of functions* $f_N : N \to N'$ *and* $f_E : E \to E'$ *preserving source and target (*$src' \circ f_E = f_N \circ src$ *and* $tar' \circ f_E = f_N \circ tar$*).*

Definition 2 (Typed Graph). *Given a graph* TG*, a* TG*-typed graph* $\langle G, tp_G \rangle$ *is a graph* G *equipped with a structure-preserving graph morphism* $tp_G : G \to TG$*. We call* TG type graph *and* $\langle G, tp_G \rangle$ instance graph over TG*. The category of* TG*-typed instance graphs is called* $\mathbf{Graph}_{TG}$*.*

The graphs we use are directed and unlabeled; for the sake of clarity, nodes (and edges) can be named by unique identifiers. Type graphs can be represented by *UML class diagrams* and instance graphs by *UML object diagrams* [19]. The typing morphism tp_G is depicted by referencing the type names. As an example, Figure 1(a) shows the type graph of the business-level style we have defined in [4]. Figure 1(b) shows a corresponding instance graph.



(a) type graph (b) instance graph

Fig. 1. Type graph and exemplary instance graph of the business-level style

According to this type graph, architectures consist of ComponentInstances which externalize their functionalities through Ports. They can interact with each other through a Connection between their Ports. The state of a communication is encoded by Request and Response message nodes.

Besides the elements for run-time configurations, the type graph also defines nodes for the application-specific types of these elements. For example, Component, PortType, and Connector nodes can be used to describe certain types of components, ports, or connections; PortTypes are characterized by provided and required Interfaces. This way, a corresponding instance graph incorporates both the actual configuration at a certain run-time state as well as application-specific type information about the involved entities.

For example, the instance graph in Fig. 1(b) defines a system that consists of an instance a of component A and an instance b of component B. Both component instances own a port of type A-Port and B-Port respectively, which could be connected by an instance of the AB-Connector. The A-Port provides the interface Int with the operation op, while the B-Port requires this interface.

Along with the type graph comes a set C of *constraints* that further restricts the set of valid instance graphs. Simple constraints already included in the class diagram are cardinalities that restrict the multiplicity of links between the elements (omitted cardinality means 0..n by default). More complex restrictions can be defined, e.g., using expressions of the *Object Constraint Language (OCL)*, which is part of the UML.

Graph transformation. Graph transformation rules [13] are used to define rewriting operations on graphs. Since our instance graphs represent system configurations, transformation rules nicely fit to define *reconfiguration operations* provided by the platform. If we encode communication-related information into the graphs, as done by the Message node and its subtypes in Fig. 1(a), then transformation rules are also suitable to represent *communication mechanisms*. A certain reconfiguration and communication scenario can be modeled as a sequence of transformation rules which are applied to an initial instance graph. The set of meaningful sequences can be restricted by additional *control processes* as discussed in Section 4.

Formally, a graph transformation rule $r : L \rightsquigarrow R$ consists of a pair of TG-typed instance graphs L, R such that the intersection $L \cap R$ is well-defined (this means that, e.g., edges which appear in both L and R are connected to the same vertices in both graphs, or that vertices with the same name have to have the same type, etc.). The left-hand side L represents the pre-conditions of the rule while the right-hand side R describes the post-conditions. The left-hand side can also state negative pre-conditions (*negative application conditions, NAG*).

According to the *Double-Pushout* semantics (DPO [14]), the application of a rule r is performed in three steps, yielding a transformation step $G \Rightarrow H$:

1. Find an occurrence o_L of the left-hand side L in the current object graph G. Formally, this is a total graph morphism $o_L : L \rightarrow G$ which maps the left-hand side L to a matching subgraph in G. The occurrence is only valid, if $o_L(L)$ cannot be extended by the forbidden elements of a NAG.